Road sign classification - EITP40

Team Members: Fabian Rosén & Joel Bengs

Overview of the Project

This project centers on crafting an intelligent traffic sign recognition model that can run on Arduino devices. Starting from the GTSRB roadsign dataset, we create seven distinct datasets for training, validation, and testing. A Convolutional Neural Network was trained (in Kaggle) and then used to process data, creating 64-feature representations compatible with the limited memory on the Arduino. Small Multi-layer Perceptrons were trained on the device and evaluated both against a high-quality dataset and images captured with the Arduino's own camera.

Results showcased successful single-device training and improved performance in distributed training. Challenges arose from low-quality Arduino-captured images. In summary, the project successfully demonstrated machine learning integration on Arduino for practical traffic sign recognition and the benefit of distributed learning.

Plan, Tasks, and Responsibilities

I. Task 1: Data Collection (Week 47, Contributor: Joel & Fabian)

The data used in the project was the Kaggle dataset *GTSRB - German Traffic Sign Recognition Benchmark*. The reason behind choosing this dataset was because of its large size and great quality, containing more than 50,000 images in total sorted into 43 classes with a good structure. That being said, the images in the dataset are real-life images and some of them are rather difficult to classify even for a human.

Figure 1: The 43 classes of the original GTSRB dataset.

To simulate the actual use of driving with the Arduino, the on-device camera was used to take pictures of road signs displayed on a computer screen. This provided a set of pictures that resemble the sub-optimal quality of pictures possible to capture with the help of the Arduino device in real life.

Data creation flow

Our dataset creation flow is presented in overview in *Figure 2*, and in detail in *Figure 3*. In summary, seven distinct datasets were created from the original GTSRB dataset. They are completely disjoint concerning samples. They can be split into three groups; A, B, and C.

- Data in group A was used for training a convolutional neural network (CNN) in Kaggle, later referred to as Model A. Dataset A was further split into training and validation sets.
- Data in group B was used for training and evaluation of on-device models. These datasets were run through the CNN part of Model A for feature extraction before being transferred to the Arduino.
- Data in group C was displayed on a screen and photographed with the Arduino's on-device camera. Images were processed to create a second testing dataset.

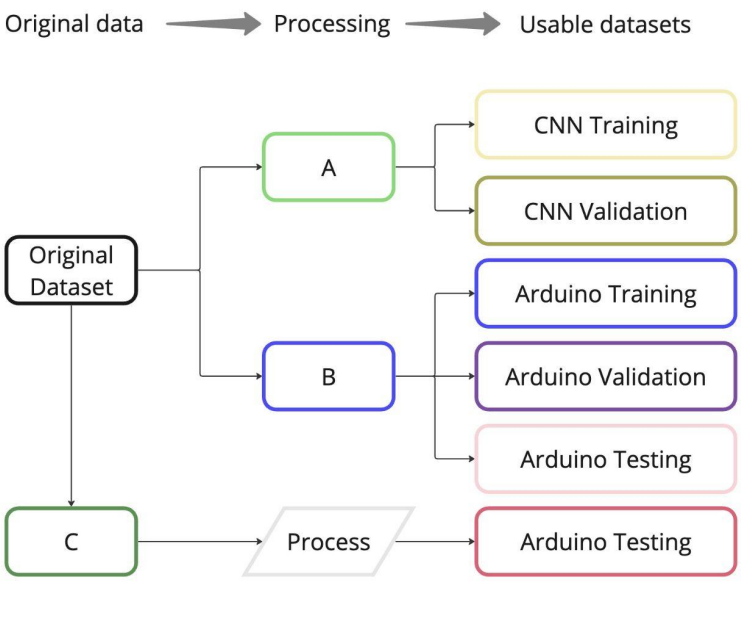

Figure 2: A simplified overview of the dataset creation process

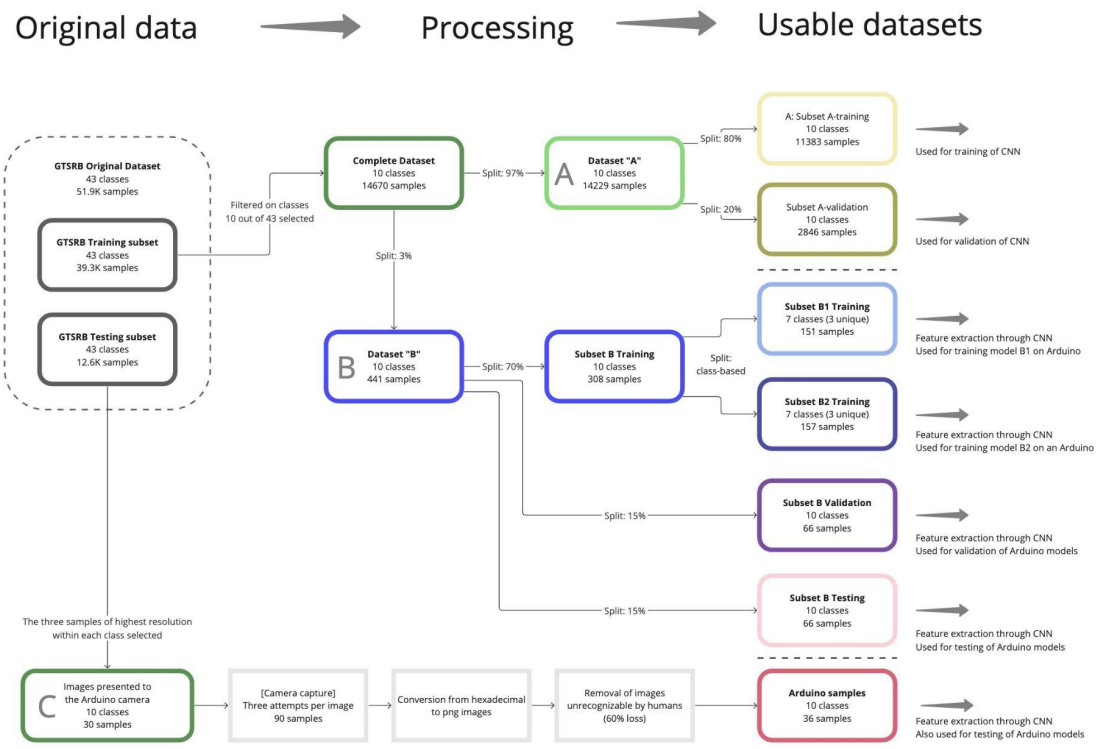

Figure 3: A detailed view of the dataset creation process.

The detailed dataset creation process in *Figure 3* outlines splitting strategies throughout. Note that only 3% of the available data is reserved for group B - this is due to the limited memory on-device. We attempted to use 10% of the data (>1400 samples) for group B, but that caused stability issues in training.

Distribution-preserving datasets

All data splits except for one are distribution-preserving, achieved using the StratifiedShuffleSplit library from scikit-learn. We implemented this to more closely resemble real-life machine learning projects, even though we could have created uniform datasets thanks to the abundance of data. *Figure 4a-b* contains histograms for all seven output datasets, along with histograms of the original dataset, dataset A and dataset B. Note that the distributions are preserved while the sizes shrink for each split.

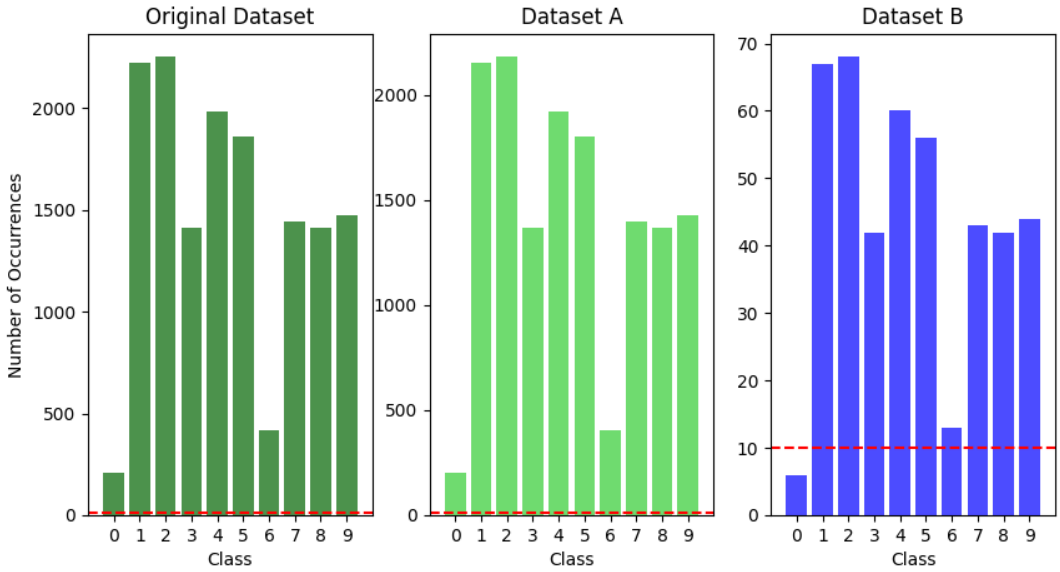

Figure 4a: Distribution of classes in datasets is being preserved after splitting the Original Dataset → A + B.

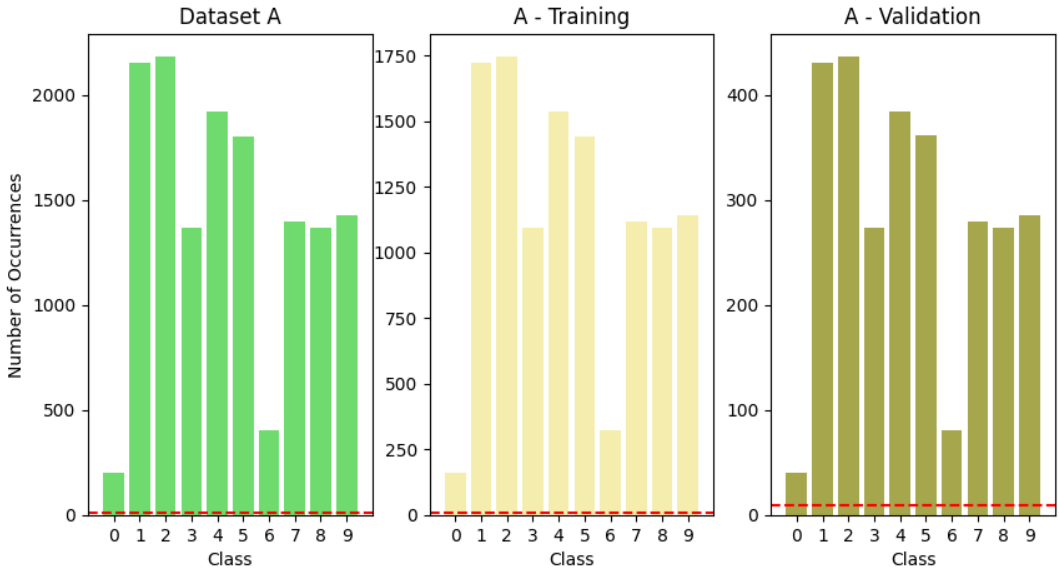

Figure 4b: Distribution of classes in datasets are being preserved after splitting A → A-Training + A-Validation.

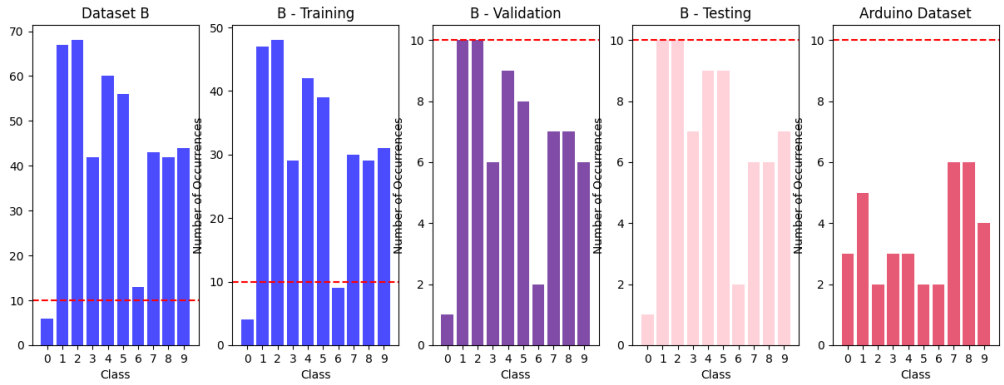

Figure 4c: Distribution of classes in datasets are being preserved after splitting B → B-Training + B-validation + B-Testing. Note that the Arduino dataset does not have a unique distribution due to being created through an entirely different process.

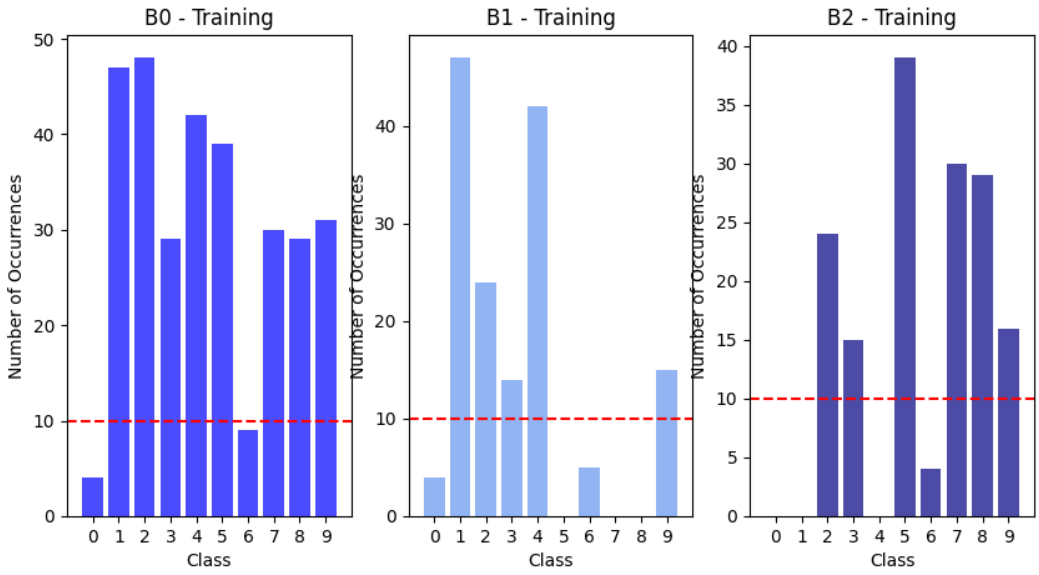

Figure 4d: This splitting of B0 → B1 + B2 was not distribution-preserving by design. Note that classes {0,1,4} are unique to dataset B1 while classes {5,7,8} are unique to dataset B2.

Disjoint training sets for distributed learning

Splitting of the training set B0 into B1 and B2 is done to facilitate distributed learning. The goal was for the two sets to be partially disjoint concerning classes: 3 classes are unique to B1, 3 classes are unique to B2, and 4 classes are present in both sets, as per *Figure 5.* This was achieved: note the absence of some classes in the respective histograms in *Figure 4c*.

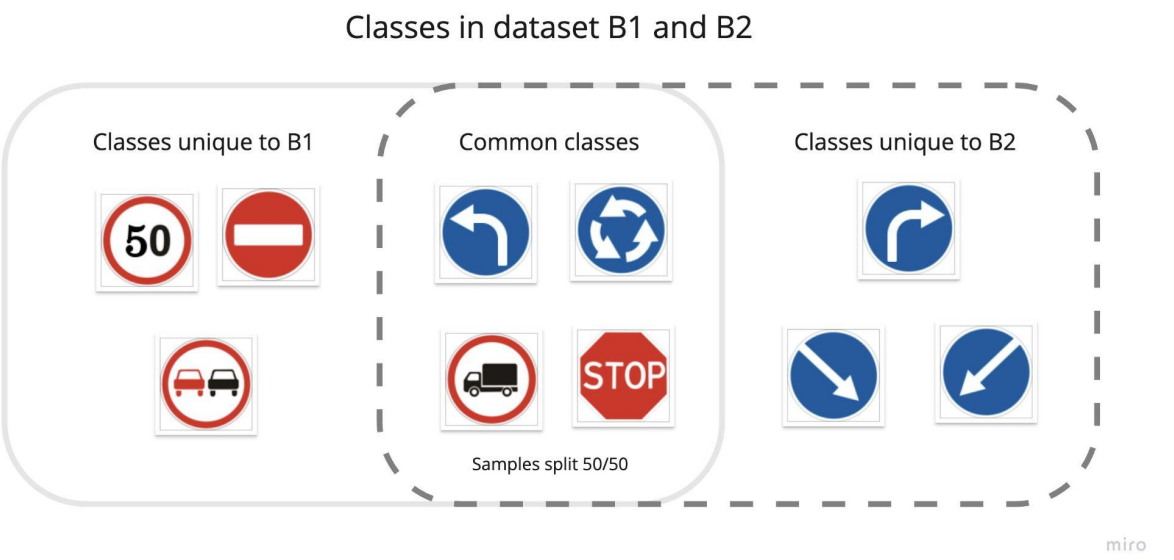

Figure 5: Classes in dataset B1 and B2

Samples captured with the Arduino on-device camera

As per the detailed view in *Figure 3*, a set of samples were shown on a screen and photographed with the on-device camera on the Arduino. This produced a hexadecimal code, which was converted to .png images using the Pillow library in Python. To extract the hexadecimal code was a very tedious and error-prone process. 60% of images were of so low quality that they had no resemblance with the original sample (as judged by a human), so they had to be discarded. The result was a set of 36 sample images, which were fed through the same automated Python pipeline as the other samples. The use case of this dataset was to further test the on-device models.

II. Task 2: ML Algorithm and Implementation on IoT Sensor (Week 48, Contributor: Fabian & Joel)

The various parts of the project were implemented either in the Kaggle environment with Python or in the Arduino IDE with C++. All data preparation, as well as construction, training and use of Model A (the CNN) was done in Kaggle. All on-device machine learning was naturally performed on the Arduino through its IDE.

The Convolutional Neural Network

Having produced the seven datasets mentioned above, a CNN (Model A) was constructed and trained using TensorFlow. Its architecture, presented in Figure 6, is an adaption of existing architecture that has been shown to perform well on this dataset (in a non-resource-constrained setting). Our modifications were to reduce the size of several filters so that the output of the CNN was a 64-feature representation of the image (as opposed to something much larger).

```
model = tf.keras.models.Sequential([
    keras.layers.Conv2D(filters=8, kernel_size=(3,3),
                        activation='relu',
                        input_shape=(30,30,3)),
    keras.layers.Conv2D(filters=16, kernel_size=(3,3),
                        activation='relu'),
    keras.layers.MaxPool2D(pool_size=(2, 2)),
    keras.layers.BatchNormalization(axis=-1),

    keras.layers.Conv2D(filters=16, kernel_size=(3,3),
                        activation='relu'),
    keras.layers.Conv2D(filters=16, kernel_size=(3,3),
                        activation='relu'),
    keras.layers.MaxPool2D(pool_size=(4, 4)),
    keras.layers.BatchNormalization(axis=-1),

    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(rate=0.5),

    keras.layers.Dense(10, activation='softmax')
])

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 8)         224

conv2d_1 (Conv2D)            (None, 26, 26, 16)        1168

max_pooling2d (MaxPooling2   (None, 13, 13, 16)        0
D)

batch_normalization (Batch   (None, 13, 13, 16)        64
Normalization)

conv2d_2 (Conv2D)            (None, 11, 11, 16)        2320

conv2d_3 (Conv2D)            (None, 9, 9, 16)          2320

max_pooling2d_1 (MaxPoolin   (None, 2, 2, 16)          0
g2D)

batch_normalization_1 (Bat   (None, 2, 2, 16)          64
chNormalization)

flatten (Flatten)            (None, 64)                0

dense (Dense)                (None, 128)               8320

batch_normalization_2 (Bat   (None, 128)               512
chNormalization)

dense_1 (Dense)              (None, 128)               16512

batch_normalization_3 (Bat   (None, 128)               512
chNormalization)

dropout (Dropout)            (None, 128)               0

dense_2 (Dense)              (None, 10)                1290

=================================================================
Total params: 33306 (130.10 KB)
Trainable params: 32730 (127.85 KB)
Non-trainable params: 576 (2.25 KB)
_________________________________________________________________
```

Figure 6a: The Architecture of Model A. Note that the CNN part produces a 64-feature output.

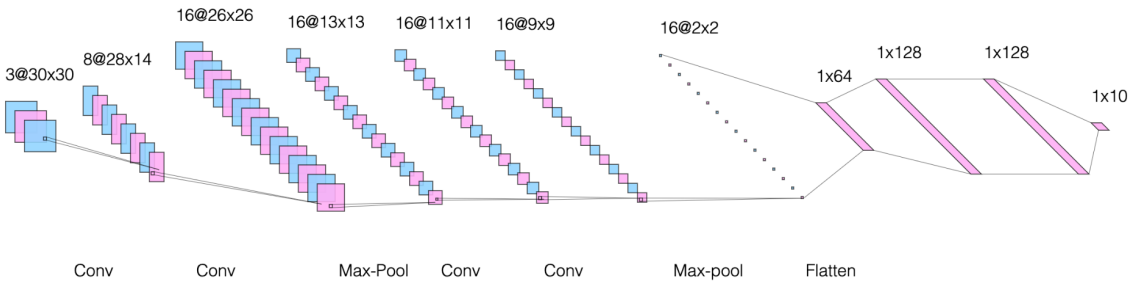

Figure 6b: Visualization of Model A, with RGB images of resolution 30x30 pixels as input.

As per *Figure 6,* Model A has four convolutional filters, with 8, 16, 16, and 16 kernels respectively. Every second filter has MaxPooling and batch normalization, and ReLu is used as an activation function for all filters to improve training speed thanks to its easy-to-compute derivative. The CNN feeds its output into an MLP with two hidden dense layers of 128 nodes each. The model's output is, as usual for classification tasks, ten nodes with softmax activation. The only type of regularization used is dropout in the last hidden layer of the MLP. More regularization could have been used, but there was no need as Model A is only used for data processing and not part of the project's main goal.

Model A was trained for 100 epochs using Stochastic Gradient Descent with a learning rate of 0.001 (as a result of a grid search) and a batch size of 32 samples (a heritage from other successful models on the same dataset). It achieved 99% accuracy in training and validation (indicating possible over-training), however, generalization performance was never tested due to the need to limit the project's scope.

Model A contains an MLP substantially larger than those trained on-device later. Its purpose was to facilitate the training of a high-quality CNN. Afterward, it was discarded. All datasets that were destined for the Arduino were passed through the CNN part of Model A. This was automated and created the desired 64-feature float representations, which is a more suitable input format than 30x30 pixel RGB images for the Arduino, due to the much smaller size.

All NumPy arrays were converted into formats suitable for C++ header files. Since many such data processing pipelines were run throughout the project, this was automated as callable functions in Python.

When implementing Arduino algorithms (as described further below), several of the algorithms were also implemented in Python in parallel. This provided a benchmark for what to expect performance-wise, facilitating easier debugging on the Arduino.

On-device Multi-layer perceptron

Since we were only two persons without knowledge in C++ in this project aimed at groups of 3 to 4 persons with C++ knowledge, we relied heavily on the C++ provided by the teacher assistants for executing ML on the Arduino devices. For the isolated ML done on only one device, we modified the code provided on the course's Canvas page as "*BP_2023*" to fit our data and use case. For the distributed ML done on two devices, we modified code from a group from the previous year to fit our project, as recommended by the teacher assistants.

The provided code for ML on one device (*BP_2023*) implements a neural network with feed-forward functionality and backpropagation (for non-stochastic gradient descent). The provided code for distributed ML on two devices builds on *BP_2023* with the addition being implemented Bluetooth functionality and the capability to share weights.

The complete workflow is visualized in *Figure 7.* Datasets are passed through the CNN, converted to 64 feature representations, and then passed on through the Arduino's model.

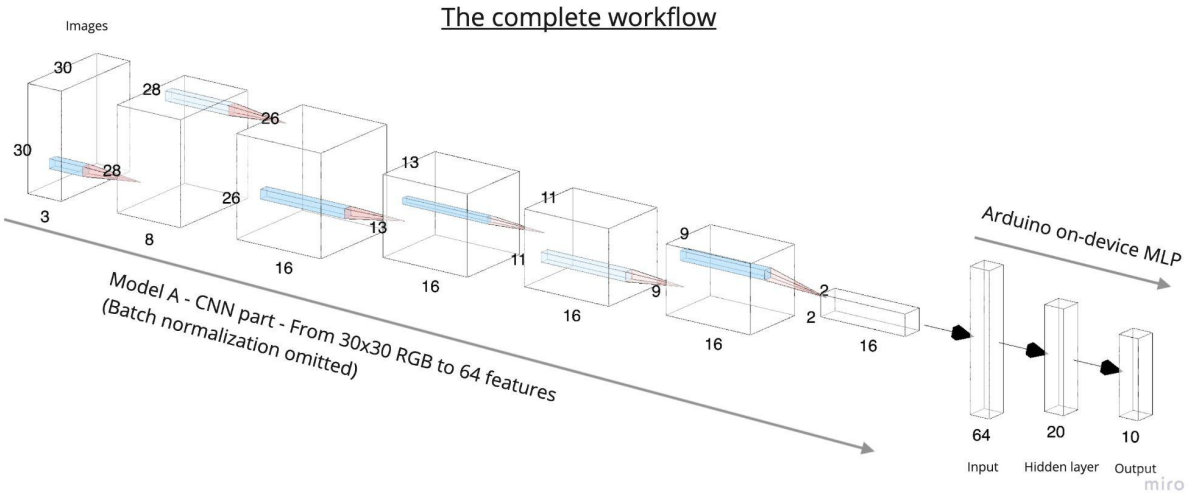

Figure 7: The complete workflow, a merger of the CNN part of Model A and any B-model on the Arduino.

III. Task 3: Evaluation and Results (Week 49, Contributor: Joel & Fabian)

ML on one device only

The results from the ML performed on one device only are presented in *Figure 8*. Training was very successful when only done on one Arduino device, with test accuracy quickly reaching 90%.

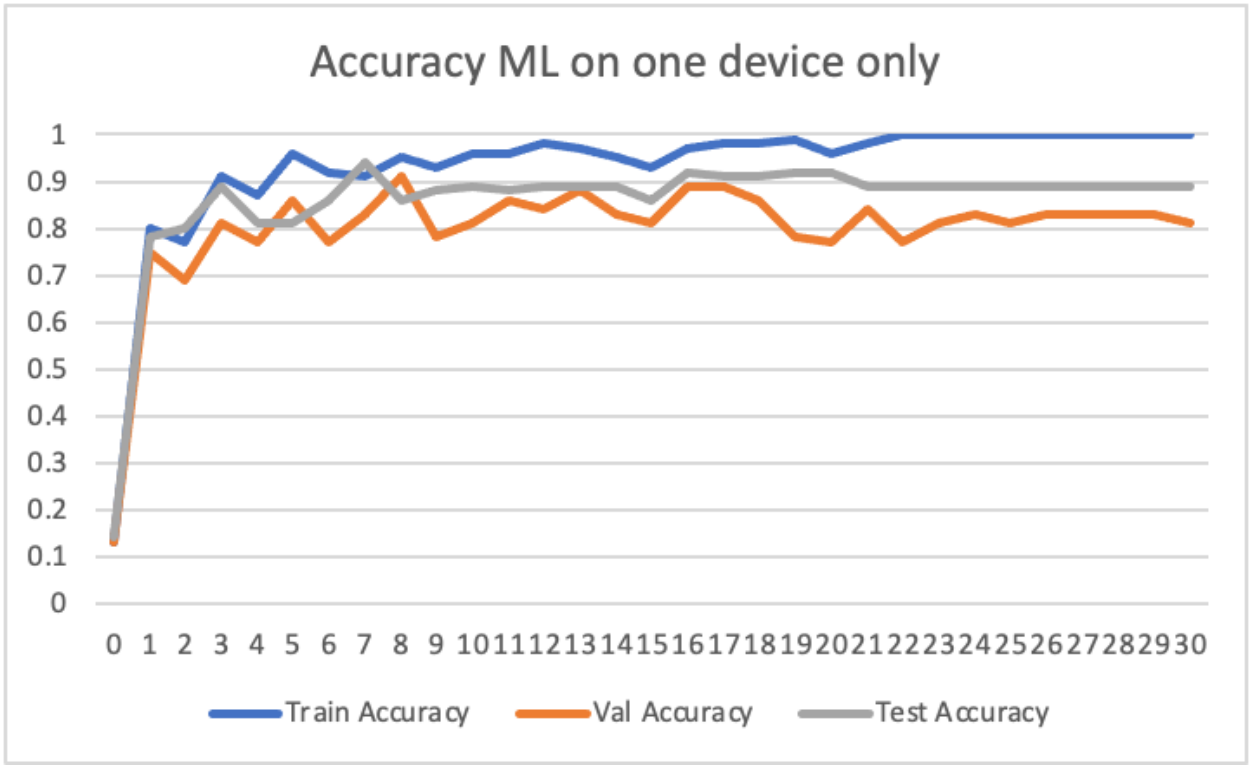

Figure 8: Accuracy over number of epochs for training on one device with full training set

Distributed ML on two devices

The summarised results from the distributed and isolated ML are presented in *Figure 9*. The training was as explained above done on training sets including 7 out of 10 classes, with 4 classes being shared and 3 classes being exclusive for each Arduino. The test set contained images from all 10 classes. As clarified in *Figure 10*, the devices perform better when sharing weight, i.e. when using distributed training.

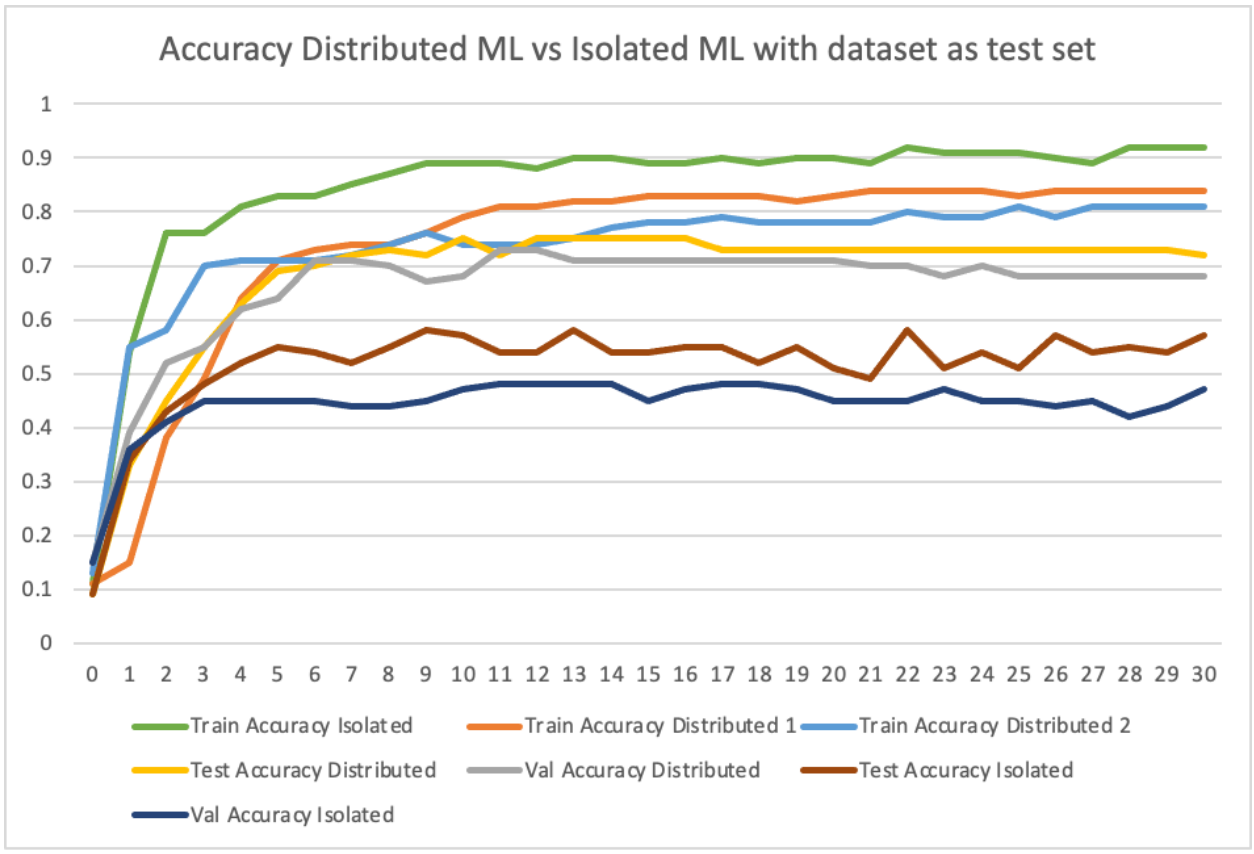

Figure 9: All accuracies from distributed and isolated training with partial training sets

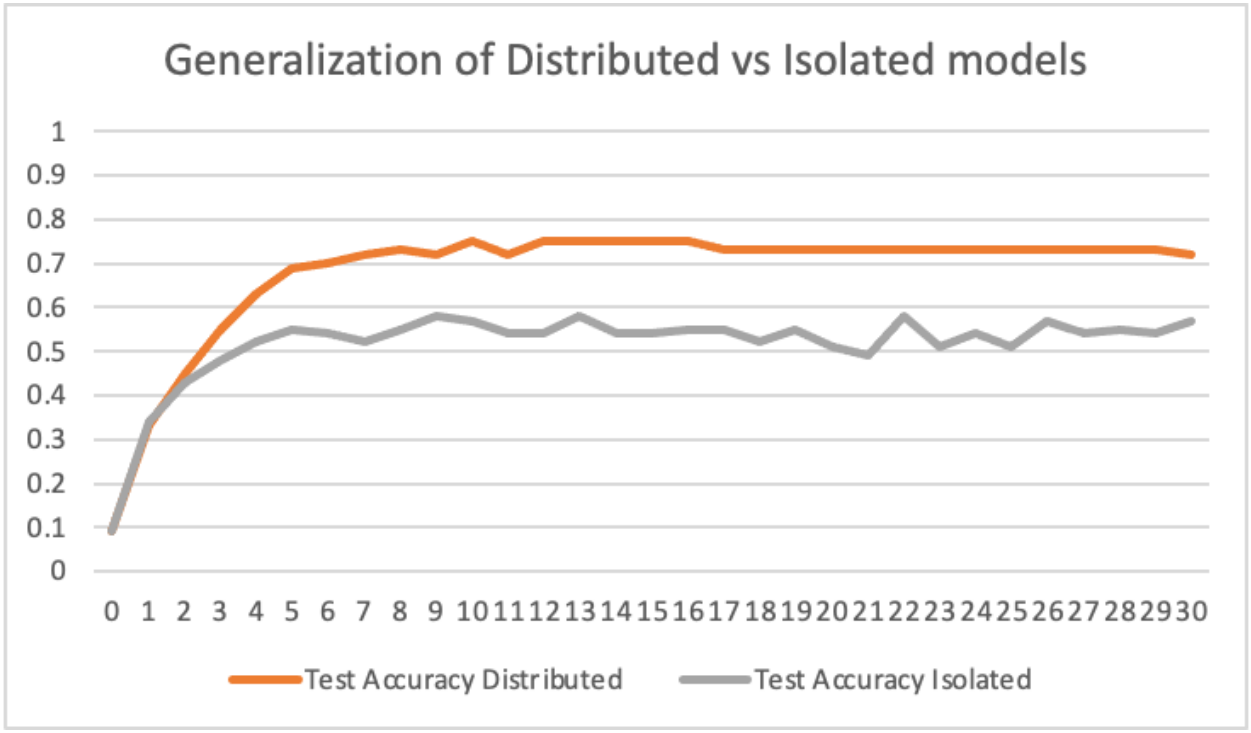

Figure 10: Test accuracy from distributed and isolated training with partial training sets

When using the pictures taken by the Arduino device as the test set, the classification of these failed completely, as can be seen in *Figure 11*. The hypothesis for this being the extremely low quality of the pictures taken by the arduino, as shown in *Figure 12* and explained in the section *Deviations from the Initial Plan*.

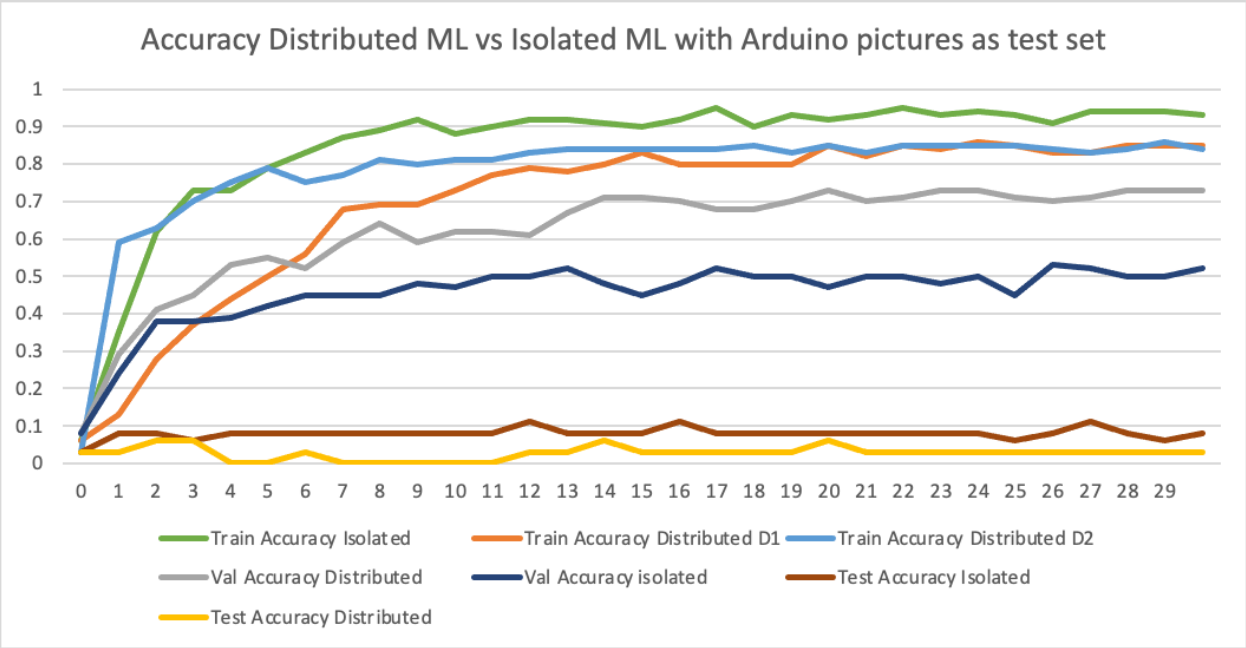

Figure 11: All accuracies from distributed and isolated training with partial training sets

Deviations from the Initial Plan

The initial plan was to be able to do everything end-to-end on the Arduino device. This means having a seamless process from taking the picture on the Arduino, processing the image and classifying it from that image. Upon realizing the memory and processing limitations of the Arduino, we had to deviate from this initial plan. The solution was to split the model into two parts, part A which was executed on a laptop, and part B which was executed on the Arduinos. Part A on the laptop was a Convolutional Neural Network (CNN) responsible for extracting 64 attributes from each picture. Part B on the Arudinos was a Multilayer Perceptron (MLP) responsible for classifying each array of 64 attributes into one of the 10 classes, being the 10 different road signs.

The first deviation from the initial plan included taking pictures of road signs with the Arduino as a test set to prove its end-to-end capability with a laptop in the middle of this process. However, because of the extremely low quality of the pictures the Arduino took, as exemplified in *Figure 12*, this test set proved to be impossible for the model to classify. Further processing of these images might have improved this, for example doing contrast normalization over both training data and test data.

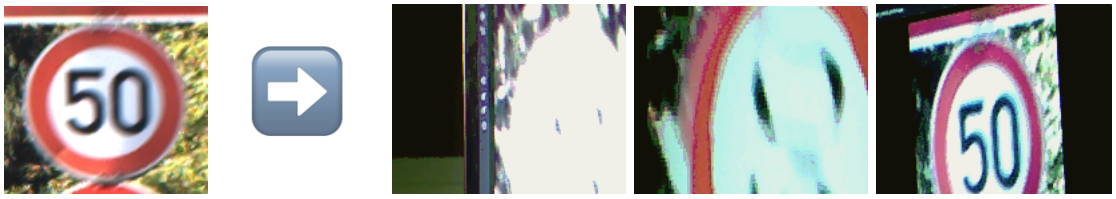

Figure 12: Sample images from the Arduino camera process. The first picture is the sample that was displayed on the screen. The next three pictures are the outcomes of three attempted photographs. Naturally, the first and second attempts have to be discarded as useless. Across the 90 attempted samples, 60% were lost.

Future research could be to capture a complete dataset using the Arduino, capturing enough pictures for training, validation, and testing, with the purpose of training completely on these pictures. Training the Arduino on pictures taken by it will make it better at classifying these. However, collecting this dataset would be a very time-consuming task, and with the questionable quality of the Arduino camera, one should consider using a camera with a quality more similar to cameras expected to perform tasks like this in real-life applications.